# Merging og hashing

# Mål

Målet for disse slides er at beskrive nogle algoritmer og datastrukturer relateret til at gemme og hente data effektivt.

Dette emne er et uddrag af kurset *DM507 Algoritmer og datastrukturer* (2. semester).

# Tilgang til data

To udbredte metoder for at tilgå data:

- Sekventiel tilgang
- Random access: tilgang via ID (også kaldet key, nøgle).

# API for sekventiel tilgang

(API = Application Programming Interface: samling af metoder).

# API for sekventiel tilgang

(API = Application Programming Interface: samling af metoder).

Operationer for læsning:

`readNext()`, `isEndOfFile()`, `open()`, `close()`

# API for sekventiel tilgang

(API = Application Programming Interface: samling af metoder).

Operationer for læsning:

`readNext()`, `isEndOfFile()`, `open()`, `close()`

Typisk anvendelse:

```
file = open(filename)
while not file.isEndOfFile()
   x = file.readNext()
   (...gør noget med x...)
file.close()
```

| 7 | 2 | 3 | 4 | 9 | 4 | 7 | 2 | 8 | 1 | 6 |
|---|---|---|---|---|---|---|---|---|---|---|

# API for sekventiel tilgang

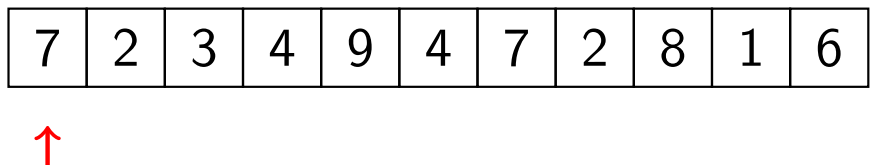
(API = Application Programming Interface: samling af metoder).

Operationer for læsning:

`readNext()`, `isEndOfFile()`, `open()`, `close()`

Typisk anvendelse:

```
file = open(filename)
while not file.isEndOfFile()
   x = file.readNext()
   (...gør noget med x...)
file.close()
```



# API for sekventiel tilgang

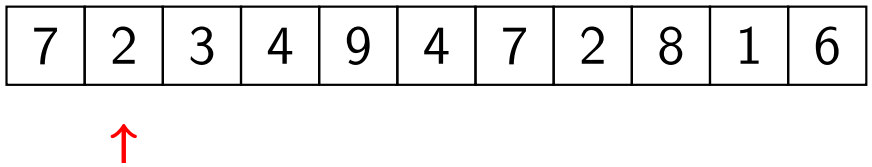
(API = Application Programming Interface: samling af metoder).

Operationer for læsning:

`readNext()`, `isEndOfFile()`, `open()`, `close()`

Typisk anvendelse:

```
file = open(filename)
while not file.isEndOfFile()
   x = file.readNext()
   (...gør noget med x...)
file.close()
```



# API for sekventiel tilgang

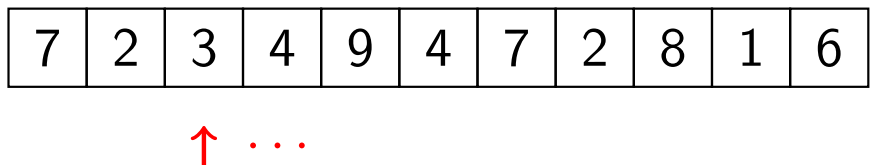
(API = Application Programming Interface: samling af metoder).

Operationer for læsning:

`readNext()`, `isEndOfFile()`, `open()`, `close()`

Typisk anvendelse:

```
file = open(filename)
while not file.isEndOfFile()
   x = file.readNext()
   (...gør noget med x...)
file.close()
```



# API for sekventiel tilgang

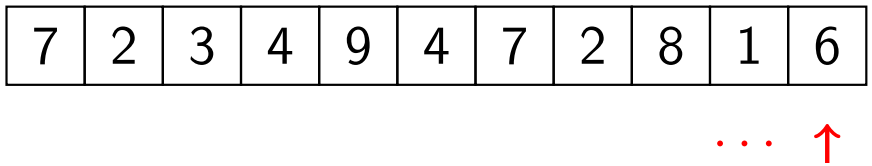
(API = Application Programming Interface: samling af metoder).

Operationer for læsning:

`readNext()`, `isEndOfFile()`, `open()`, `close()`

Typisk anvendelse:

```
file = open(filename)
while not file.isEndOfFile()
   x = file.readNext()
   (...gør noget med x...)
file.close()
```



# API for sekventiel tilgang

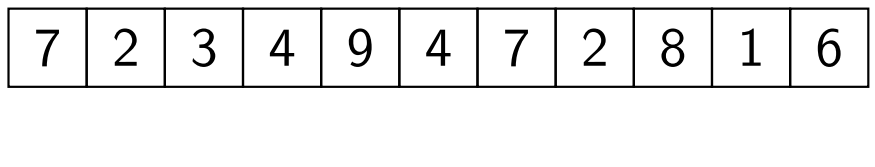
(API = Application Programming Interface: samling af metoder).

Operationer for læsning:

`readNext()`, `isEndOfFile()`, `open()`, `close()`

Typisk anvendelse:

```
file = open(filename)
while not file.isEndOfFile()
   x = file.readNext()
   (...gør noget med x...)
file.close()
```



# API for sekventiel tilgang

Operationer for skrivning:

Operationer: `writeNext()`, `open()`, `close()`

# API for sekventiel tilgang

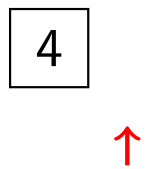
Operationer for skrivning:

Operationer: `writeNext()`, `open()`, `close()`

# API for sekventiel tilgang

Operationer for skrivning:

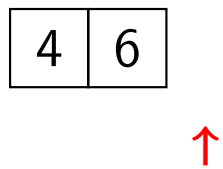
Operationer: `writeNext()`, `open()`, `close()`



# API for sekventiel tilgang

Operationer for skrivning:

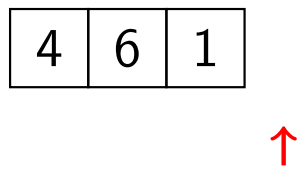
Operationer: `writeNext()`, `open()`, `close()`



# API for sekventiel tilgang

Operationer for skrivning:

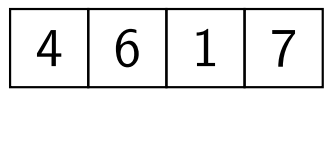
Operationer: `writeNext()`, `open()`, `close()`



# API for sekventiel tilgang

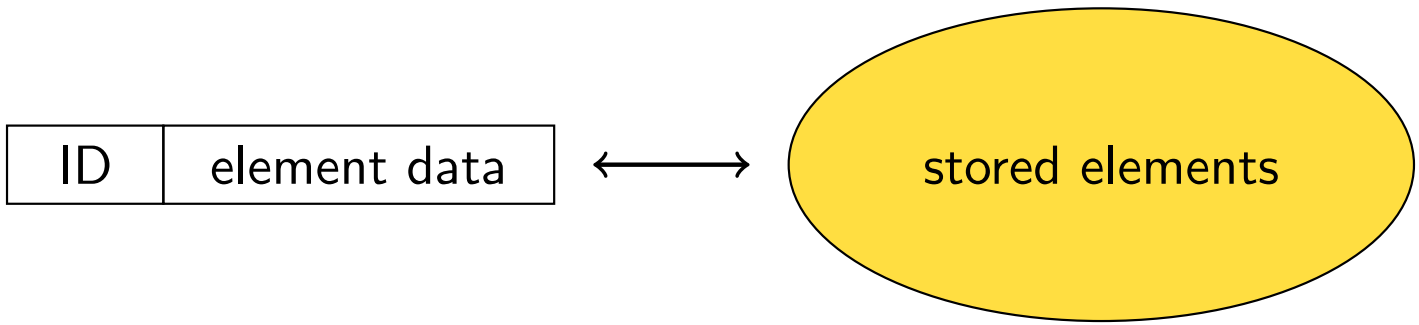
Operationer for skrivning:

Operationer: `writeNext()`, `open()`, `close()`



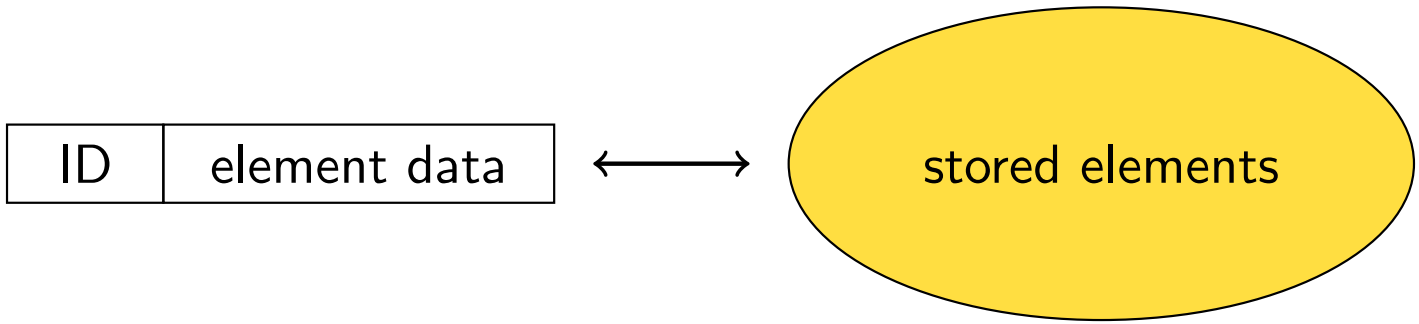
# API for random access tilgang

Tilgang via ID (også kaldet key, nøgle) for elementer.



# API for random access tilgang

Tilgang via ID (også kaldet key, nøgle) for elementer.




Operationer:

- `findElm(ID)` (returnerer element data)
- `insertElm(ID,elementData)`
- `deleteElm(ID)`
- `open()`, `close()`

# API for random access tilgang

```
findElm(ID)
insertElm(ID,elementData)
deleteElm(ID)
open(), close()
```

Eksempler:

# API for random access tilgang

```
findElm(ID)
insertElm(ID,elementData)
deleteElm(ID)
open(), close()
```

Eksempler:

- Databaser (ID = CPR, f.eks.)

# API for random access tilgang

```
findElm(ID)
insertElm(ID,elementData)
deleteElm(ID)
open(), close()
```

Eksempler:

- Databaser (ID = CPR, f.eks.)
- Dictionary i Python

```
data = dict[ID]
dict[ID] = data
del dict[ID]
```

# API for random access tilgang

```
findElm(ID)
insertElm(ID,elementData)
deleteElm(ID)
open(), close()
```

Eksempler:

- Databaser (ID = CPR, f.eks.)
- Dictionary i Python

```
data = dict[ID]
dict[ID] = data
del dict[ID]
```

- Lister i Python/arrays i Java kan ses som specialtilfælde hvor ID = index

```
data = list[7]
list[7] = data
list[7] = None
```

# Dagens spørgsmål

1. Hvad kan det simple API Sekventiel tilgang bruges til?

# Dagens spørgsmål

1. Hvad kan det simple API Sekventiel tilgang bruges til?
2. Hvordan implementeres det mere avancerede API Random access tilgang ?

# Dagens spørgsmål

1. Hvad kan det simple API Sekventiel tilgang bruges til?
2. Hvordan implementeres det mere avancerede API Random access tilgang ?

Bemærk: Alle datakilder kan tilgås med sekventiel tilgang. (Og for nogle kan kun sekventiel tilgang laves effektivt.)

- Harddisk
- CD
- Bånd
- Streaming over net
- Data genereret on-the-fly af et andet program
- Data i et array (liste i Python)

# Hvad kan laves med sekventiel tilgang?

# Hvad kan laves med sekventiel tilgang?

F.eks.:

- Lineær søgning

# Hvad kan laves med sekventiel tilgang?

F.eks.:

- Lineær søgning
- Find største element

# Hvad kan laves med sekventiel tilgang?

F.eks.:

- Lineær søgning
- Find største element
- Find sum og antal af elementer (og dermed gennemsnit)

# Hvad kan laves med sekventiel tilgang?

F.eks.:

- Lineær søgning
- Find største element
- Find sum og antal af elementer (og dermed gennemsnit)
- Slå to sorterede lister sammen til én sorteret liste

# Hvad kan laves med sekventiel tilgang?

F.eks.:

- Lineær søgning
- Find største element
- Find sum og antal af elementer (og dermed gennemsnit)
- Slå to sorterede lister sammen til én sorteret liste
- Mergesort

# Hvad kan laves med sekventiel tilgang?

F.eks.:

- Lineær søgning
- Find største element
- Find sum og antal af elementer (og dermed gennemsnit)
- Slå to sorterede lister sammen til én sorteret liste
- Mergesort
- Generaliseret merge (eksempler senere)

# Hvad kan laves med sekventiel tilgang?

F.eks.:

- Lineær søgning
- Find største element
- Find sum og antal af elementer (og dermed gennemsnit)
- Slå to sorterede lister sammen til én sorteret liste
- Mergesort
- Generaliseret merge (eksempler senere)
- Bubblesort, selectionsort, insertionsort, quicksort (ikke pensum i DM534)

# Hvad kan laves med sekventiel tilgang?

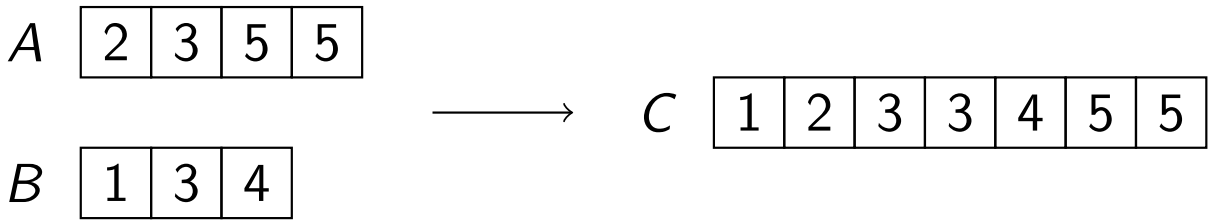
F.eks.:

- Lineær søgning
- Find største element
- Find sum og antal af elementer (og dermed gennemsnit)
- Slå to sorterede lister sammen til én sorteret liste
- Mergesort
- Generaliseret merge (eksempler senere)
- Bubblesort, selectionsort, insertionsort, quicksort (ikke pensum i DM534)

Morale: Man kan lave forbavsende mange algoritmer baseret på det simple API Sekventiel tilgang.

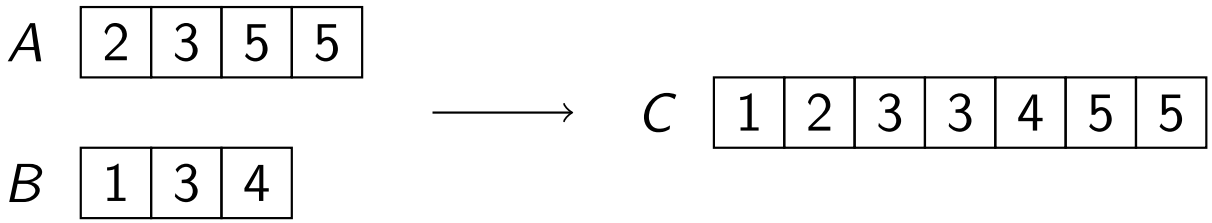
# Merge

Mål: Slå to (stigende) sorterede lister $A$ og $B$ sammen til én sorteret liste $C$.



# Merge

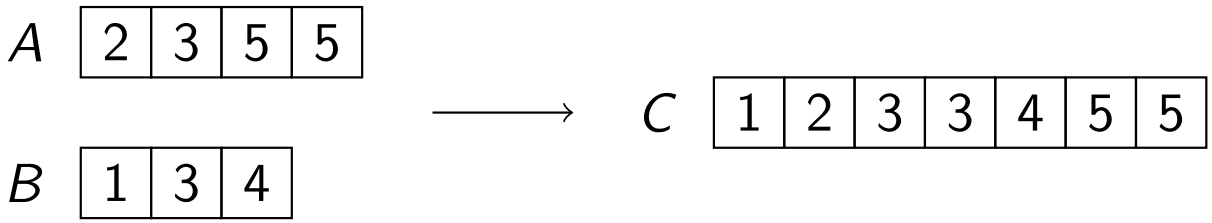
Mål: Slå to (stigende) sorterede lister $A$ og $B$ sammen til én sorteret liste $C$.




Et merge-skridt: Sammenlign (nuværende) forreste element i $A$ og $B$, og flyt det mindste af disse over sidst i $C$.

# Merge

Mål: Slå to (stigende) sorterede lister $A$ og $B$ sammen til én sorteret liste $C$.

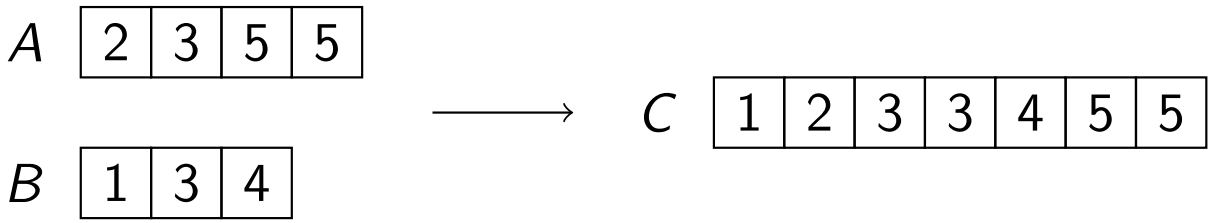



Et merge-skridt: Sammenlign (nuværende) forreste element i $A$ og $B$, og flyt det mindste af disse over sidst i $C$.

Start med tom $C$
Sålænge både $A$ og $B$ er ikke-tomme:
    Udfør et merge-skridt
Hvis enten $A$ eller $B$ er ikke-tom:
    Flyt resten af dens elementer over sidst i $C$

# Merge

Mål: Slå to (stigende) sorterede lister $A$ og $B$ sammen til én sorteret liste $C$.



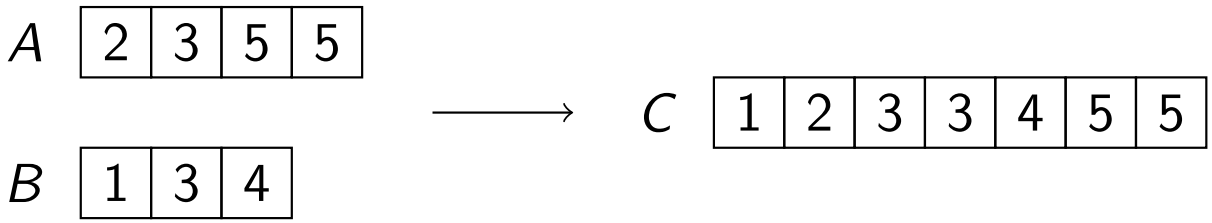

Et merge-skridt: Sammenlign (nuværende) forreste element i $A$ og $B$, og flyt det mindste af disse over sidst i $C$.

Start med tom $C$
Sålænge både $A$ og $B$ er ikke-tomme:
    Udfør et merge-skridt
Hvis enten $A$ eller $B$ er ikke-tom:
    Flyt resten af dens elementer over sidst i $C$

Er det en korrekt metode (dvs. er $C$ ved afslutning sorteret)?

# Merge

Mål: Slå to (stigende) sorterede lister $A$ og $B$ sammen til én sorteret liste $C$.




Et merge-skridt: Sammenlign (nuværende) forreste element i $A$ og $B$, og flyt det mindste af disse over sidst i $C$.

Start med tom $C$
Sålænge både $A$ og $B$ er ikke-tomme:
    Udfør et merge-skridt
Hvis enten $A$ eller $B$ er ikke-tom:
    Flyt resten af dens elementer over sidst i $C$

Er det en korrekt metode (dvs. er $C$ ved afslutning sorteret)?

Korrekthed kan ses ved hjælp af en invariant.

# Korrekthed og køretid for merge

Mål: Slå to (stigende) sorterede lister $A$ og $B$ sammen til én sorteret liste $C$.

Et merge-skridt: Sammenlign (nuværende) forreste element i $A$ og $B$, og flyt det mindste af disse over sidst i $C$.

Start med tom $C$
Sålænge både $A$ og $B$ er ikke-tomme:
    Udfør et merge-skridt
Hvis enten $A$ eller $B$ er ikke-tom:
    Flyt resten af dens elementer over sidst i $C$

# Korrekthed og køretid for merge

Mål: Slå to (stigende) sorterede lister $A$ og $B$ sammen til én sorteret liste $C$.

Et merge-skridt: Sammenlign (nuværende) forreste element i $A$ og $B$, og flyt det mindste af disse over sidst i $C$.

> Start med tom $C$
> Sålænge både $A$ og $B$ er ikke-tomme:
>  Udfør et merge-skridt
> Hvis enten $A$ eller $B$ er ikke-tom:
>  Flyt resten af dens elementer over sidst i $C$

Korrekthed (at $C$ ved afslutning er sorteret) kan ses ved flg. invariant:

> Nuværende version af $A$, $B$, $C$ er sorterede, og ingen elementer i $A$, $B$ er mindre end noget element i $C$.

# Korrekthed og køretid for merge

Mål: Slå to (stigende) sorterede lister $A$ og $B$ sammen til én sorteret liste $C$.

Et merge-skridt: Sammenlign (nuværende) forreste element i $A$ og $B$, og flyt det mindste af disse over sidst i $C$.

> Start med tom $C$
> Sålænge både $A$ og $B$ er ikke-tomme:
>  Udfør et merge-skridt
> Hvis enten $A$ eller $B$ er ikke-tom:
>  Flyt resten af dens elementer over sidst i $C$

Korrekthed (at $C$ ved afslutning er sorteret) kan ses ved flg. invariant:

> Nuværende version af $A$, $B$, $C$ er sorterede, og ingen elementer i $A$, $B$ er mindre end noget element i $C$.

Tid:

# Korrekthed og køretid for merge

Mål: Slå to (stigende) sorterede lister $A$ og $B$ sammen til én sorteret liste $C$.

Et merge-skridt: Sammenlign (nuværende) forreste element i $A$ og $B$, og flyt det mindste af disse over sidst i $C$.

> Start med tom $C$
> Sålænge både $A$ og $B$ er ikke-tomme:
>  Udfør et merge-skridt
> Hvis enten $A$ eller $B$ er ikke-tom:
>  Flyt resten af dens elementer over sidst i $C$

Korrekthed (at $C$ ved afslutning er sorteret) kan ses ved flg. invariant:

> Nuværende version af $A$, $B$, $C$ er sorterede, og ingen elementer i $A$, $B$ er mindre end noget element i $C$.

Tid: $\Theta(|A| + |B|)$

# Mergesort

Merge lister af længde 1 (disse er automatisk sorterede) parvis sammen til sorterede lister af længde 2 ($= 2^1$). Arbejdet svarer til at røre alle elementer én gang, dvs. er $\Theta(n)$.

Derefter merge de sorterede lister af længde 2 til sorterede lister af længde 4 ($= 2^2$). Arbejdet svarer til at røre alle elementer én gang, dvs. er $\Theta(n)$.

Derefter længde 4 til længde 8 ($= 2^3$),...

Efter $i$ skridt: længde $= 2^i$.

Indtil længde $n$ nås, dvs. alle elementer er i én sorteret liste. Dvs. vi har sorteret elementerne.

# Mergesort

Merge lister af længde 1 (disse er automatisk sorterede) parvis sammen til sorterede lister af længde 2 ($= 2^1$). Arbejdet svarer til at røre alle elementer én gang, dvs. er $\Theta(n)$.

Derefter merge de sorterede lister af længde 2 til sorterede lister af længde 4 ($= 2^2$). Arbejdet svarer til at røre alle elementer én gang, dvs. er $\Theta(n)$.

Derefter længde 4 til længde 8 ($= 2^3$),...

Efter $i$ skridt: længde $= 2^i$.

Indtil længde $n$ nås, dvs. alle elementer er i én sorteret liste. Dvs. vi har sorteret elementerne.

Der er lavet $\log n$ runder når længde $n$ nås [da $2^i = n \Leftrightarrow i = \log n$ ].

# Mergesort

Merge lister af længde 1 (disse er automatisk sorterede) parvis sammen til sorterede lister af længde 2 ($= 2^1$). Arbejdet svarer til at røre alle elementer én gang, dvs. er $\Theta(n)$.

Derefter merge de sorterede lister af længde 2 til sorterede lister af længde 4 ($= 2^2$). Arbejdet svarer til at røre alle elementer én gang, dvs. er $\Theta(n)$.

Derefter længde 4 til længde 8 ($= 2^3$),... | Efter $i$ skridt: længde $= 2^i$.

Indtil længde $n$ nås, dvs. alle elementer er i én sorteret liste. Dvs. vi har sorteret elementerne.

Der er lavet $\log n$ runder når længde $n$ nås [da $2^i = n \Leftrightarrow i = \log n$ ].

Tid i alt:

# Mergesort

Merge lister af længde 1 (disse er automatisk sorterede) parvis sammen til sorterede lister af længde 2 ($= 2^1$). Arbejdet svarer til at røre alle elementer én gang, dvs. er $\Theta(n)$.

Derefter merge de sorterede lister af længde 2 til sorterede lister af længde 4 ($= 2^2$). Arbejdet svarer til at røre alle elementer én gang, dvs. er $\Theta(n)$.

Derefter længde 4 til længde 8 ($= 2^3$),... $\boxed{\text{Efter } i \text{ skridt: længde} = 2^i.}$

Indtil længde $n$ nås, dvs. alle elementer er i én sorteret liste. Dvs. vi har sorteret elementerne.

Der er lavet $\log n$ runder når længde $n$ nås [da $2^i = n \Leftrightarrow i = \log n$ ].

Tid i alt: $\boxed{\Theta(n \log n)}$ [da vi $\log n$ gange laver $\Theta(n)$ arbejde].

# Mergesort

Merge lister af længde 1 (disse er automatisk sorterede) parvis sammen til sorterede lister af længde 2 ($= 2^1$). Arbejdet svarer til at røre alle elementer én gang, dvs. er $\Theta(n)$.

Derefter merge de sorterede lister af længde 2 til sorterede lister af længde 4 ($= 2^2$). Arbejdet svarer til at røre alle elementer én gang, dvs. er $\Theta(n)$.

Derefter længde 4 til længde 8 ($= 2^3$),... $\boxed{\text{Efter } i \text{ skridt: længde} = 2^i.}$

Indtil længde $n$ nås, dvs. alle elementer er i én sorteret liste. Dvs. vi har sorteret elementerne.

Der er lavet $\log n$ runder når længde $n$ nås [da $2^i = n \Leftrightarrow i = \log n$ ].

Tid i alt: $\boxed{\Theta(n \log n)}$ [da vi $\log n$ gange laver $\Theta(n)$ arbejde].

Se illustration af algoritmen og dens analyse næste side.

# Mergesort

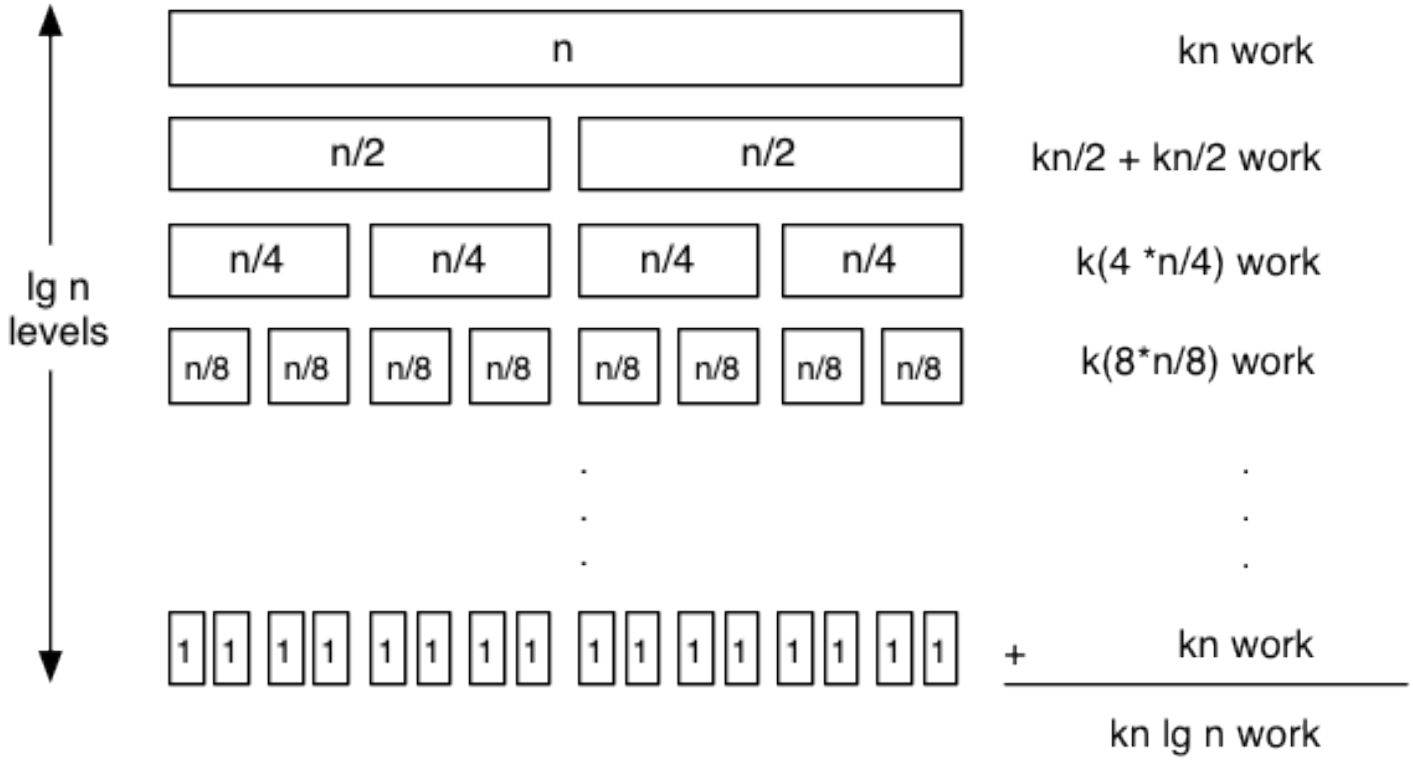



Figure: Andrew Myers, Cornell University

# Generaliseret merge

Under merge af to sorterede lister $A$ og $B$ vil elementer med samme værdi "møde" hinanden under merge-skridtene. Dette kan udnyttes:

# Generaliseret merge

Under merge af to sorterede lister $A$ og $B$ vil elementer med samme værdi "møde" hinanden under merge-skridtene. Dette kan udnyttes:

Eksempel: $A$ er datafil med (ID,data)-elementer, sorteret efter ID, $B$ er liste af (ID,opdatering)-elementer som angiver opdateringer, sorteret efter ID.

At gennemløbe $A$ og $B$ via merge-skridt vil tillade os at opdatere data i $A$. NB: de to lister behøver ikke være lige lange.

# Generaliseret merge

Under merge af to sorterede lister $A$ og $B$ vil elementer med samme værdi "møde" hinanden under merge-skridtene. Dette kan udnyttes:

Eksempel: $A$ er datafil med (ID,data)-elementer, sorteret efter ID, $B$ er liste af (ID,opdatering)-elementer som angiver opdateringer, sorteret efter ID.

At gennemløbe $A$ og $B$ via merge-skridt vil tillade os at opdatere data i $A$. NB: de to lister behøver ikke være lige lange.

Eksempel: To lister $A$ og $B$ repræsenterer mængder (og er derfor hver især uden dubletter).

At gennemløbe $A$ og $B$ via merge-skridt vil tillade os at generere f.eks. $A \cap B$: Hvis et merge-step ser to ens elementer som forreste i $A$ og $B$, flyttes det ene over sidst i $C$, og den anden smides væk. Hvis et merge-step ser to forskellige, smides den mindste væk.

# Hvordan laves API Random access tilgang?

Én metode: Hashing.

# Hvordan laves API Random access tilgang?

Én metode: Hashing.

 ?

# Hvordan laves API Random access tilgang?

Én metode: Hashing.

 ?

Hash $\sim$ hacher [fransk] $\sim$ hakke i stykker.

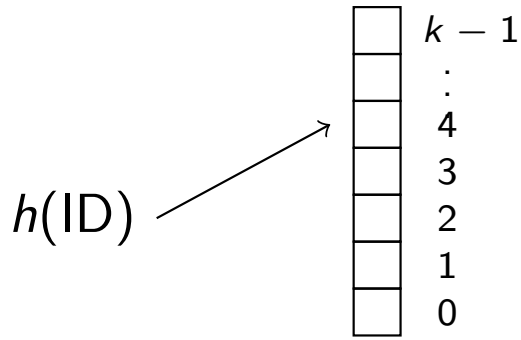
# Hashing

Idé: tildel hver ID et index i et array $A$ hvor element gemmes.
Hash-funktion $h$:

# Hashing

Idé: tildel hver ID et index i et array $A$ hvor element gemmes.
Hash-funktion $h$:

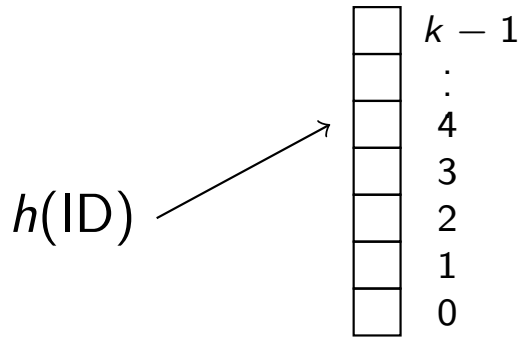
$$h(\text{ID}) = \text{index i } A$$



# Hashing

Idé: tildel hver ID et index i et array $A$ hvor element gemmes.
Hash-funktion $h$:

$$h(\text{ID}) = \text{index i } A$$




Eksempel (antag ID'er er heltal):

$$h(x) = x \bmod k$$

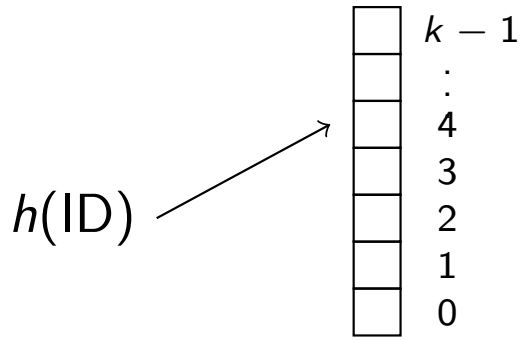
hvor $k = |A|$.

Bemærk at $h(x) \in \{0, 1, 2, \ldots, k-1\}$, så det er altid et lovligt index.

# Hashing

Idé: tildel hver ID et index i et array $A$ hvor element gemmes.
Hash-funktion $h$:

$$h(\text{ID}) = \text{index i } A$$




Eksempel (antag ID'er er heltal):

$$h(x) = x \bmod k$$

hvor $k = |A|$.

Bemærk at $h(x) \in \{0, 1, 2, \ldots, k-1\}$, så det er altid et lovligt index.

Med $k = 41$:

$$\begin{aligned} h(46) &= 5 && (\text{da } 1 \cdot 41 + 5 = 46) \\ h(12) &= 12 && (\text{da } 0 \cdot 41 + 12 = 12) \\ h(100) &= 18 && (\text{da } 2 \cdot 41 + 18 = 100) \\ h(479869) &= 5 && (\text{da } 11704 \cdot 41 + 5 = 479869) \end{aligned}$$

# Hashing

Hvorfor ikke bare

$$h(x) = x?$$

# Hashing

Hvorfor ikke bare

$$h(x) = x?$$

Eksempel: gem 5 CPR-numre.

# Hashing

Hvorfor ikke bare

$$h(x) = x?$$

Eksempel: gem 5 CPR-numre.

CPR-nummre: 180781-2345 $\in \{0, 1, 2, \ldots 10^{10} - 1\}$

Så størrelse af array $A$ skal være $10^{10}$ for at gemme 5 tal. Spild af plads ($10^{10}$ bytes = 4 Gb).

# Hashing

Hvorfor ikke bare

$$h(x) = x?$$

Eksempel: gem 5 CPR-numre.

CPR-nummre: 180781-2345 $\in \{0, 1, 2, \ldots 10^{10} - 1\}$

Så størrelse af array $A$ skal være $10^{10}$ for at gemme 5 tal. Spild af plads ($10^{10}$ bytes = 4 Gb).

Ofte er nøgler heltal (32 eller 64 bits), dvs. har $2^{32} \approx 10^{10}$ eller $2^{64} \approx 10^{20}$ muligheder. Dvs. samme situation eller værre.

# Kollisioner

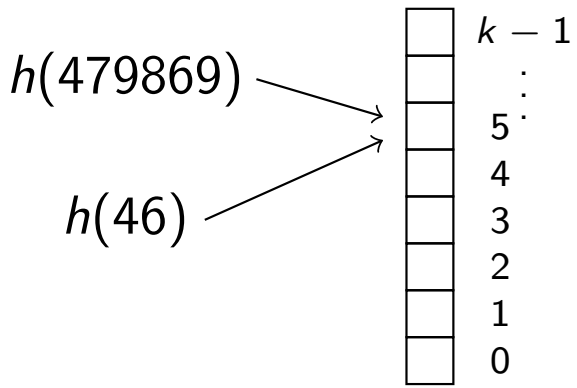
$h(x) = x \bmod 41$:

$$
\begin{aligned}
h(46) &= 5 && (\text{da } 1 \cdot 41 + 5 = 46)\\
h(12) &= 12 && (\text{da } 0 \cdot 41 + 12 = 12)\\
h(100) &= 18 && (\text{da } 2 \cdot 41 + 18 = 100)\\
h(479869) &= 5 && (\text{da } 11704 \cdot 41 + 5 = 479869)
\end{aligned}
$$

# Kollisioner

$h(x) = x \bmod 41$:

$h(46) = 5$ (da $1 \cdot 41 + 5 = 46$)
$h(12) = 12$ (da $0 \cdot 41 + 12 = 12$)
$h(100) = 18$ (da $2 \cdot 41 + 18 = 100$)
$h(479869) = 5$ (da $11704 \cdot 41 + 5 = 479869$)

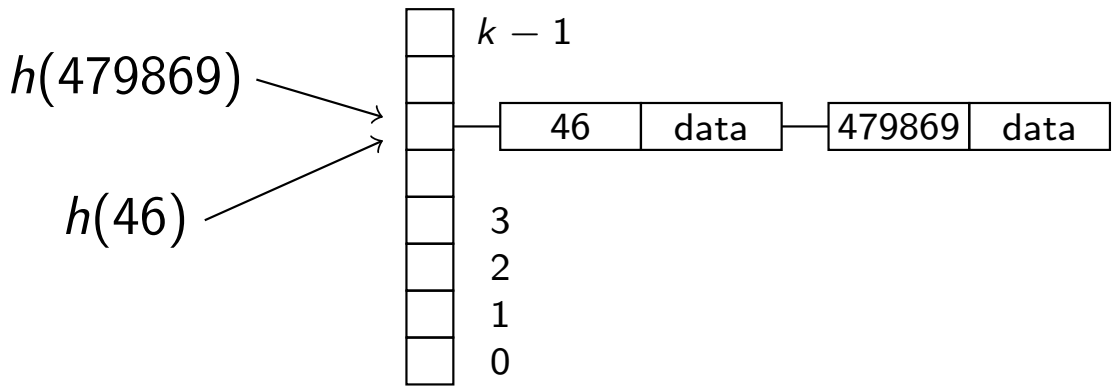


# Én løsning: Chaining

Gem alle elementer for en array-celle i en lænket liste.

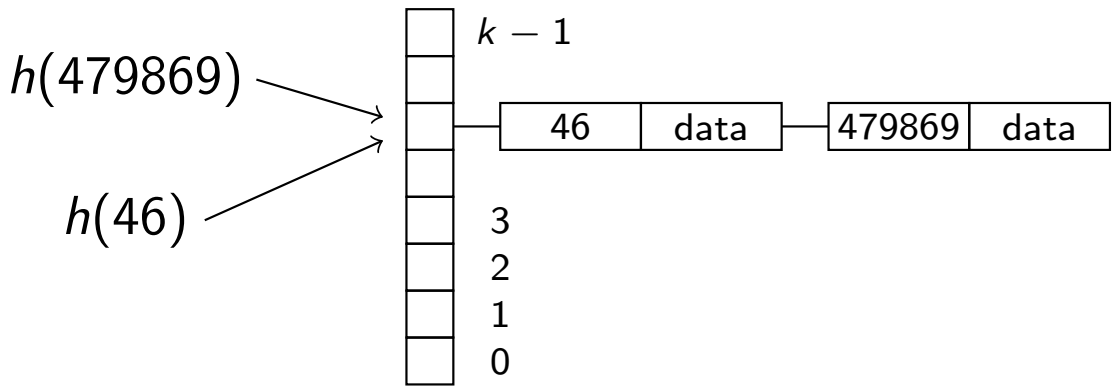
# Én løsning: Chaining

Gem alle elementer for en array-celle i en lænket liste.



# Én løsning: Chaining

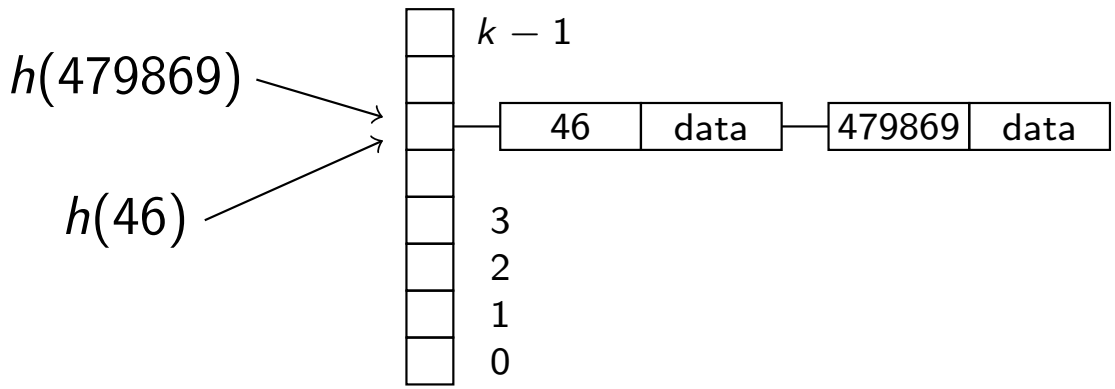
Gem alle elementer for en array-celle i en lænket liste.




De listen skal gennemløbes sekventielt, stiger tiden fra $\Theta(1)$ til $\Theta(|\text{liste}|)$.

# Én løsning: Chaining

Gem alle elementer for en array-celle i en lænket liste.




De listen skal gennemløbes sekventielt, stiger tiden fra $\Theta(1)$ til $\Theta(|\text{liste}|)$.

Vi vil derfor gerne have få kollisioner.

# Kan kollisioner undgås?

Det afhænger af forholdet mellem hash-funktionen $h$ og det konkrete datasæt, som er gemt.

# Kan kollisioner undgås?

Det afhænger af forholdet mellem hash-funktionen $h$ og det konkrete datasæt, som er gemt.

I værste fald hash'er alle $n$ elementer til samme celle. Tid: $\Theta(n)$.

# Kan kollisioner undgås?

Det afhænger af forholdet mellem hash-funktionen $h$ og det konkrete datasæt, som er gemt.

I værste fald hash'er alle $n$ elementer til samme celle. Tid: $\Theta(n)$.

Hvis $n$ (antal elementer gemt) er større end $k$ (array-størrelse), er der mindst én kollision

# Kan kollisioner undgås?

Det afhænger af forholdet mellem hash-funktionen $h$ og det konkrete datasæt, som er gemt.

I værste fald hash'er alle $n$ elementer til samme celle. Tid: $\Theta(n)$.

Hvis $n$ (antal elementer gemt) er større end $k$ (array-størrelse), er der mindst én kollision (duehulsprincippet).

# Kan kollisioner undgås?

Det afhænger af forholdet mellem hash-funktionen $h$ og det konkrete datasæt, som er gemt.

I værste fald hash'er alle $n$ elementer til samme celle. Tid: $\Theta(n)$.

Hvis $n$ (antal elementer gemt) er større end $k$ (array-størrelse), er der mindst én kollision (duehulsprincippet).

Hvad hvis vi antager $h$ "indsætter tallene i celler tilfældigt"?

NB: Dette er ikke en brugbar hash-funktion, da vi ikke kan finde elementerne igen (vi kan ikke huske, hvor de er lagt). Men man kan argumentere for, at dette giver en jævnest mulig fordeling set over alle mulige datasæt. Det giver derfor en nedre grænse for, hvor godt en hash-funktion kan opføre sig. *Vi studerer derfor nu denne situation.*

# Fødselsdage og hashing

# Fødselsdage og hashing

Situation: $n$ tilfældige mennesker går ind i et rum.

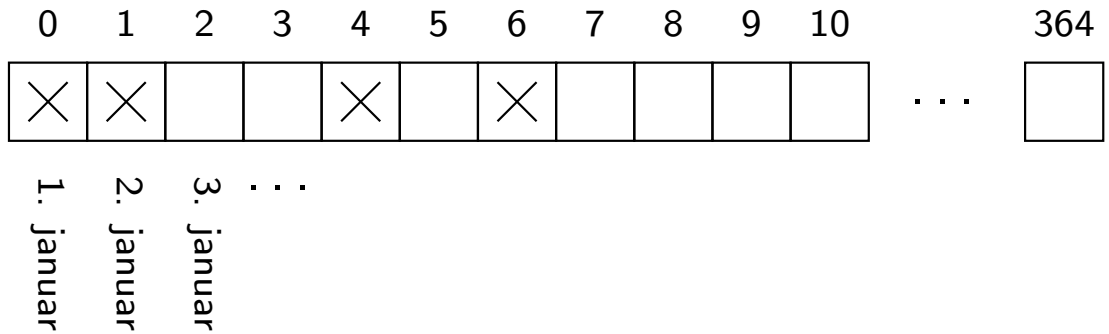
Spørgsmål: Er der nogen der har fødselsdag samme dato i året?

# Fødselsdage og hashing

Situation: $n$ tilfældige mennesker går ind i et rum.

Spørgsmål: Er der nogen der har fødselsdag samme dato i året?

Bemærk: at samle personer med tilfældige fødselsdage svarer til at indsætte tilfældigt i en tabel af størrelse 365 (= antal dage i et år).




Ovenfor er fire personer samlet (hvor ingen har fødselsdag samme dag).

*Dette er altså et eksempel på den (hashing) situation vi ønsker at studere.*

# Fødselsdage og hashing

Situation: $n$ tilfældige mennesker går ind i et rum.
Spørgsmål: Er der nogen der har fødselsdag samme dato i året?

# Fødselsdage og hashing

Situation: $n$ tilfældige mennesker går ind i et rum.
Spørgsmål: Er der nogen der har fødselsdag samme dato i året?

| $n$ | Sandsynlighed for (mindst) to med samme fødselsdag |
|---|---|
| 0 | 0 |
| 1 | 0 |
| 2 | $1/365$ |
| $\vdots$ | $\vdots$ |
| $\boxed{?}$ | $1/2$ |
| $\vdots$ | $\vdots$ |
| 366 | 1 |

Spørgsmål: for hvilket $n$ bliver sandsynligheden $\geq 1/2$?

# Fødselsdage og hashing

Bemærk:

P(nogen har samme fødseldag) = 1 - P(ingen har samme fødseldag)

# Fødselsdage og hashing

Bemærk:

P(nogen har samme fødseldag) = 1 - P(ingen har samme fødseldag)

Så for at svare på spørgsmålet kigger vi i stedet på

P(ingen med samme fødseldag blandt de $n$ første personer),

og spørger, hvornår den er mindre end $1/2$.

# Fødselsdage og hashing

Ingen med samme fødseldag blandt de $n$ første personer

$\Updownarrow$

1) Ingen med samme fødselsedag blandt de $n-1$ første personer

OG

2) den $n$'te persons fødselsdag falder ikke sammen med nogen af disses.

# Fødselsdage og hashing

Ingen med samme fødseldag blandt de $n$ første personer

$\Updownarrow$

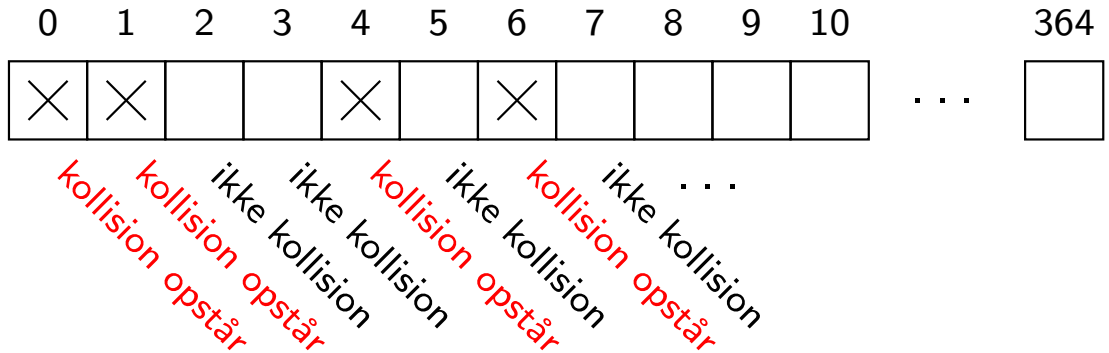
1) Ingen med samme fødselsedag blandt de $n-1$ første personer

OG

2) den $n$'te persons fødselsdag falder ikke sammen med nogen af disses.

Hvis 1) gælder, er præcis $n-1$ forskellige datoer optaget når den $n$'te person går ind i rummet. Hvis personerne antages at have tilfældige fødselsdage, er sandsynligheden for 2) derfor $(365-(n-1))/365$.

Situationen for $n=5$ (dvs. 4 personer allerede til stede, uden kollision):



# Fødselsdage og hashing

Hvis vi kalder P(ingen med samme fødseldag blandt de $n$ første personer) for $s_n$, kan vi se af ovenstående at

$$s_n = s_{n-1} \cdot \frac{365 - (n-1)}{365}$$

(Eftersom OG mellem uafhængige hændelser bliver til gange.)

Da $s_1$ naturligvis er 1 (med kun én person i rummet er der helt sikkert ingen med samme fødselsdag), ser vi at:

$$\begin{aligned}
s_1 &= 1 \\
s_2 &= s_1 \cdot \tfrac{364}{365} = 1 \cdot \tfrac{364}{365} = 0.9972\ldots \\
s_3 &= s_2 \cdot \tfrac{363}{365} = 1 \cdot \tfrac{364}{365} \cdot \tfrac{363}{365} = 0.9917\ldots \\
s_4 &= s_3 \cdot \tfrac{362}{365} = 1 \cdot \tfrac{364}{365} \cdot \tfrac{363}{365} \cdot \tfrac{362}{365} = 0.9836\ldots \\
&\vdots
\end{aligned}$$

# Fødselsdagsparadokset

Hvis man udregner disse udtryk (evt. via at skrive et program, der gør det), ses at

$$s_{22} = 0.5243\ldots$$
$$s_{23} = 0.4927\ldots$$

# Fødselsdagsparadokset

Hvis man udregner disse udtryk (evt. via at skrive et program, der gør det), ses at

$$s_{22} = 0.5243\ldots$$
$$s_{23} = 0.4927\ldots$$

Vi var interesserede i, for hvilket $n$ det var mere sandsynligt at nogen havde samme fødselsdags end at ingen havde.

Dvs. for hvilket $n$ at $s_n \leq 1/2$.

Af beregningen ovenfor får vi vores svar: allerede når $n$ bliver 23.

# Fødselsdagsparadokset

Hvis man udregner disse udtryk (evt. via at skrive et program, der gør det), ses at

$$s_{22} = 0.5243\ldots$$
$$s_{23} = 0.4927\ldots$$

Vi var interesserede i, for hvilket $n$ det var mere sandsynligt at nogen havde samme fødselsdags end at ingen havde.

Dvs. for hvilket $n$ at $s_n \leq 1/2$.

Af beregningen ovenfor får vi vores $\boxed{\text{svar}}$: allerede når $n$ bliver 23.

Navnet "fødselsdagsparadokset" bruges, fordi dette er et mindre antal personer end de fleste intuitivt vil gætte på.

# Fødselsdagsparadokset

Hvis man udregner disse udtryk (evt. via at skrive et program, der gør det), ses at

$$s_{22} = 0.5243\ldots$$
$$s_{23} = 0.4927\ldots$$

Vi var interesserede i, for hvilket $n$ det var mere sandsynligt at nogen havde samme fødselsdags end at ingen havde.

Dvs. for hvilket $n$ at $s_n \leq 1/2$.

Af beregningen ovenfor får vi vores $\boxed{\text{svar}}$: allerede når $n$ bliver 23.

Navnet "fødselsdagsparadokset" bruges, fordi dette er et mindre antal personer end de fleste intuitivt vil gætte på.

Af samme slags beregninger fås f.eks. $s_{50} = 0.0296\ldots$. Dvs. at der er under 3% sandsynlighed for, at der i en gruppe af 50 personer ikke er to med samme fødselsdato.

# Fødselsdagsparadokset og hashing

Recall: At samle personer med tilfældige fødselsdage svarer til at indsætte tilfældigt i en tabel af størrelse 365 (antal dage i et år).

Ovenstående metode kan derfor bruges til at beregne sandsynligheden for, at kollisioner opstår ved indsættelse af $n$ elementer i en hashtabel, hvis det antages at hashfunktionen tildeler tabel-indekser til elementer på en tilfældig måde. Man skal blot udskifte 365 med den aktuelle tabelstørrelse.

# Fødselsdagsparadokset og hashing

Recall: At samle personer med tilfældige fødselsdage svarer til at indsætte tilfældigt i en tabel af størrelse 365 (antal dage i et år).

Ovenstående metode kan derfor bruges til at beregne sandsynligheden for, at kollisioner opstår ved indsættelse af $n$ elementer i en hashtabel, hvis det antages at hashfunktionen tildeler tabel-indekser til elementer på en tilfældig måde. Man skal blot udskifte 365 med den aktuelle tabelstørrelse.

Morale: vi kan se af sådanne beregninger, at den første kollision opstår overraskende hurtigt, når der indsættes i hashtabeller.
Det *er* derfor nødvendigt at have et system til at klare kollisioner (f.eks. de lænkede lister fra tidligere) for at kunne bruge hashing i praksis.